# File Management

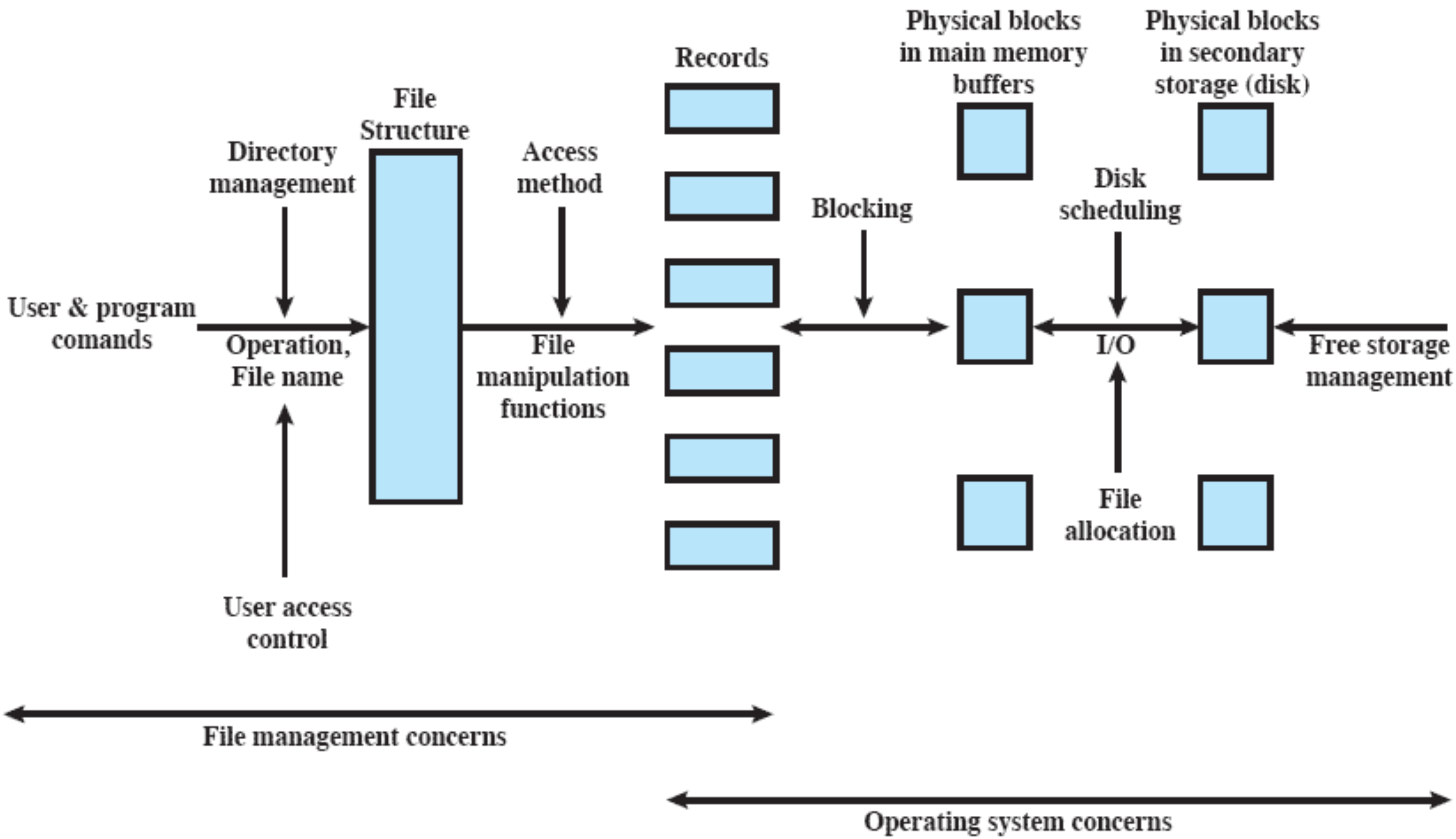
CS3026 Operating Systems

Lecture 19

# File Management Objectives

- Meet the data management needs of the user
- Guarantee that the data in the file are valid
- Optimise performance
- Provide I/O support for a variety of storage device types
- Minimise the potential for lost or destroyed data
- Provide a standardized set of I/O interface routines to user processes
- Provide I/O support for multiple users

# File Management



# Files

- Files provide a way to store information on disk
- Properties
  - Persistence / long-term existence
  - Shareable between processes
    - Have associated file permissions, attributes that express ownership, allow a controlled sharing of files
  - Organisational Structure / File System
    - Files can be organised into hierarchical structures to reflect the relationships among files

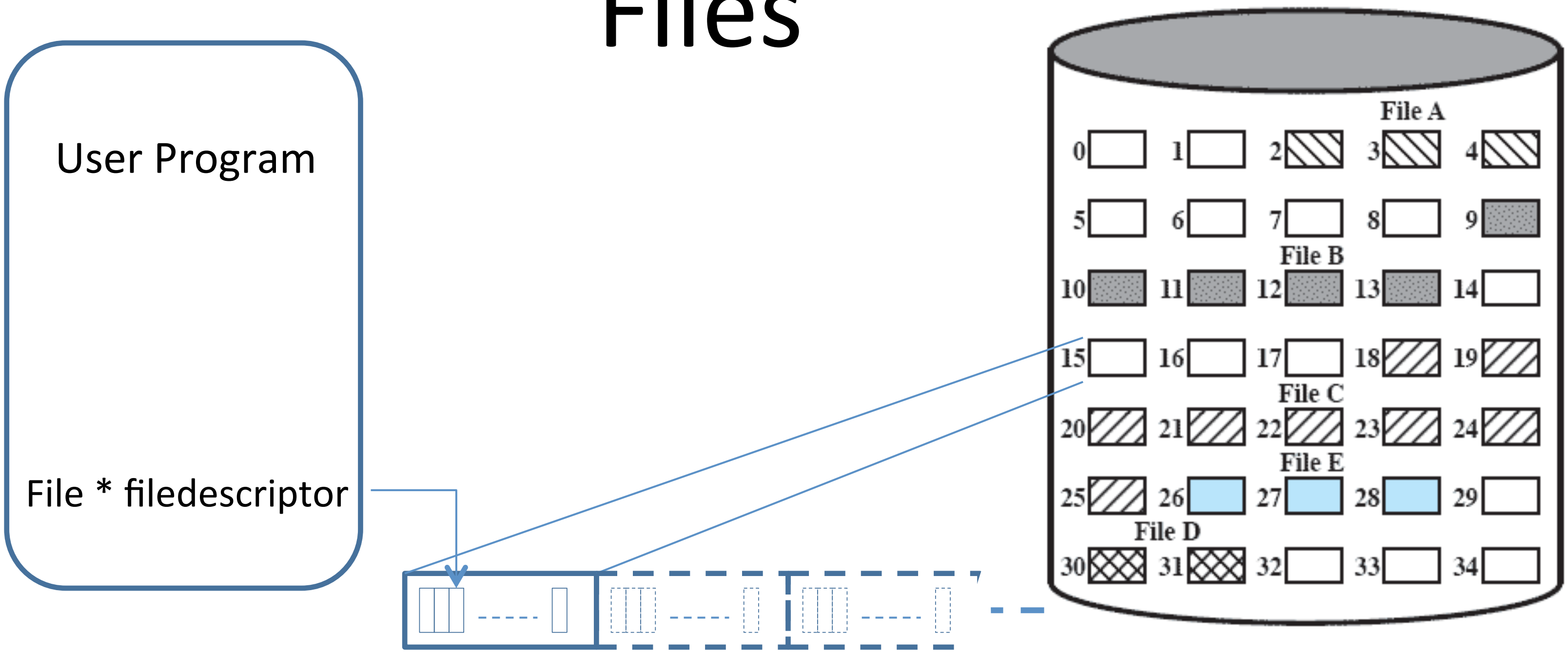
# Files

- Files are an abstraction concept
  - Users operate with a simple model of a byte stream being written to or read from disk
  - Operating system hides details how disk space is allocated to store the information represented by a file
- File systems manage files on disk space

# File Abstraction

- Operations
  - read, write, seek, create, delete
- Meta-data that describes a file
  - Directory entry stores file attributes
  - File attributes
    - Name, type
    - Location: where to find the actual data on disk
    - Size
    - Access control / protection: who may read / write /execute
    - Time: creation, last access, etc
    - Version
- Meta-data has to be stored on disk as well, usually in the form of “directory” entries

# Files




- Program regards file as a byte stream, file descriptor points to buffer
- On the hard disk, files exist as a set of disk blocks
- Operating system loads disk blocks belonging to a file into buffer
- Questions:
  - Which disk blocks belong to a file?
  - Which block is the next block in sequence?

# File Allocation, Storage Management

- The operating system / file management is responsible for allocating blocks to files
- Two issues
  - Allocated-space management: record how space on secondary storage is allocated to files
  - Free-space management: OS must keep track of the space available for allocation
- This is done by the "File System"

# File Systems

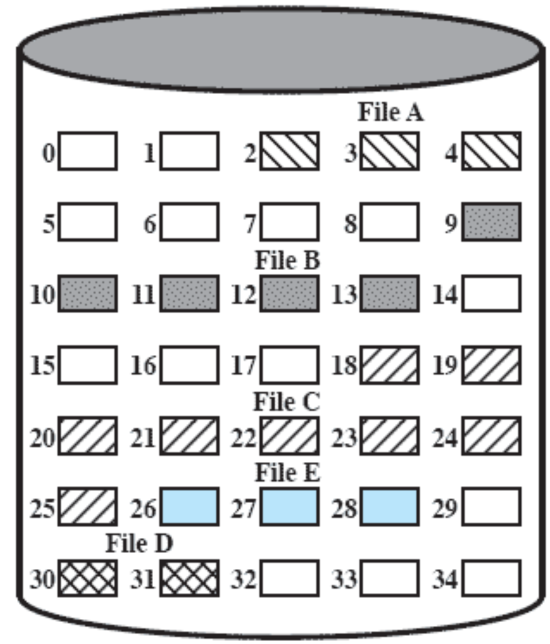
- File systems organise disk space
  - The disk itself becomes a data object – container for files
  - Organisation unit is a file – a sequence of blocks
  - Each block is a contiguous sequence of bytes, fixed block size
- Concerns:
  - Localisation: Records where and how files are stored
  - Structure: Files are organised in directories / folders
  - Access: Allows the creation of files, read and write operations
  - Performance: reduce I/O operations with buffering
  - Reliability: can recover from system crash and faults
  - Security: Protection and ownership

# File Allocation Method

- Block allocation strategies for persistent storage
  - Contiguous allocation
  - Non-contiguous allocation:
    - chained allocation
    - Indexed allocation
    - FAT, i-Nodes

# Contiguous Allocation of Blocks

- Simplest form, simple to implement, excellent read performance as a file spans across a contiguous set of disk blocks
- Over time, disk becomes fragmented, compaction necessary, external fragmentation
- Infeasible for disk management, was used on magnetic tapes
- Is again important for write-once optical devices such as CD-ROMS
  - File size is known in advance, file is written in one action, occupies a contiguous space

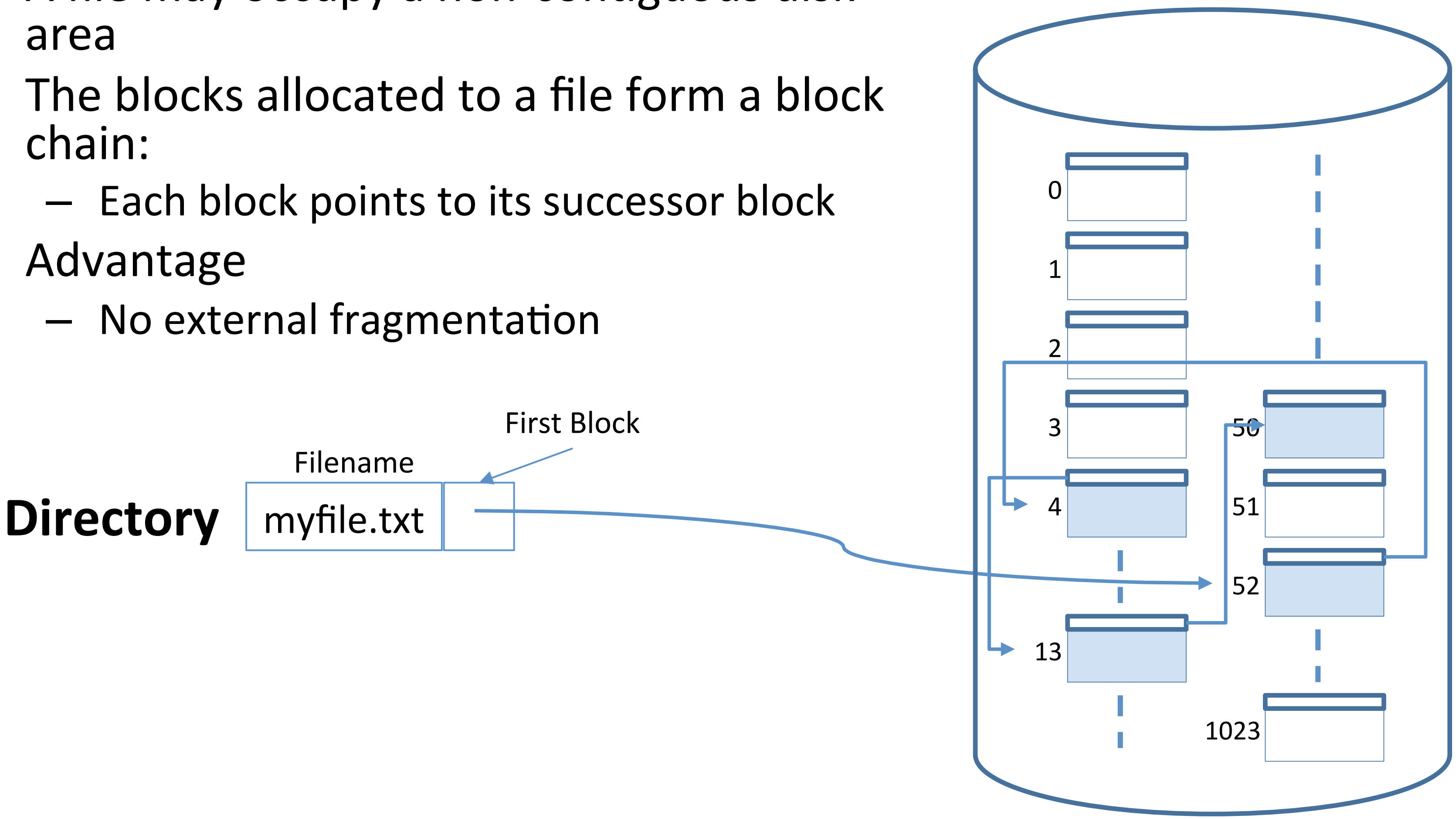



File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| File A | 2 | 3 |
| File B | 9 | 5 |
| File C | 18 | 8 |
| File D | 30 | 2 |
| File E | 26 | 3 |

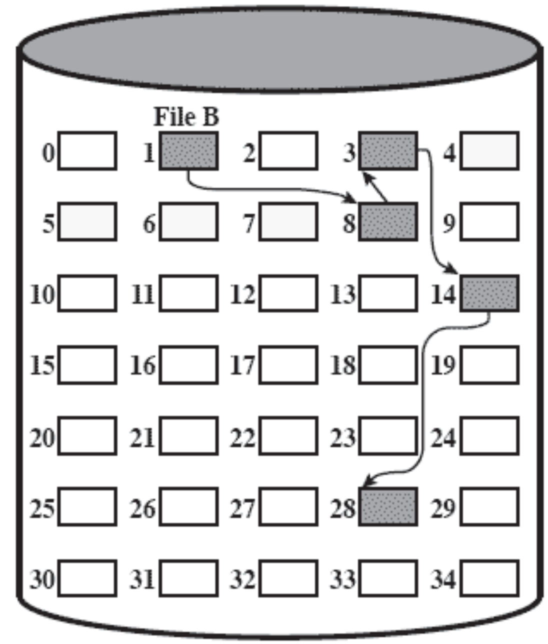
# Chained Allocation
# Non-contiguous Allocation

- A file may occupy a non-contiguous disk area
- The blocks allocated to a file form a block chain:
  - Each block points to its successor block
- Advantage
  - No external fragmentation



# Chained Allocation

- Reading a file sequentially is straight-forward
  - Follow the pointer to the next chain element
- Random access extremely slow
  - We have to follow the chain pointers until we find the right disk block
  - I/O operations for each visited block: must be read to access pointer and read next block
- Waste of memory
  - Chain pointer is part of disk block
  - A small part (32-bit or 64-bit address, 4 or 8 bytes) are wasted on these pointers

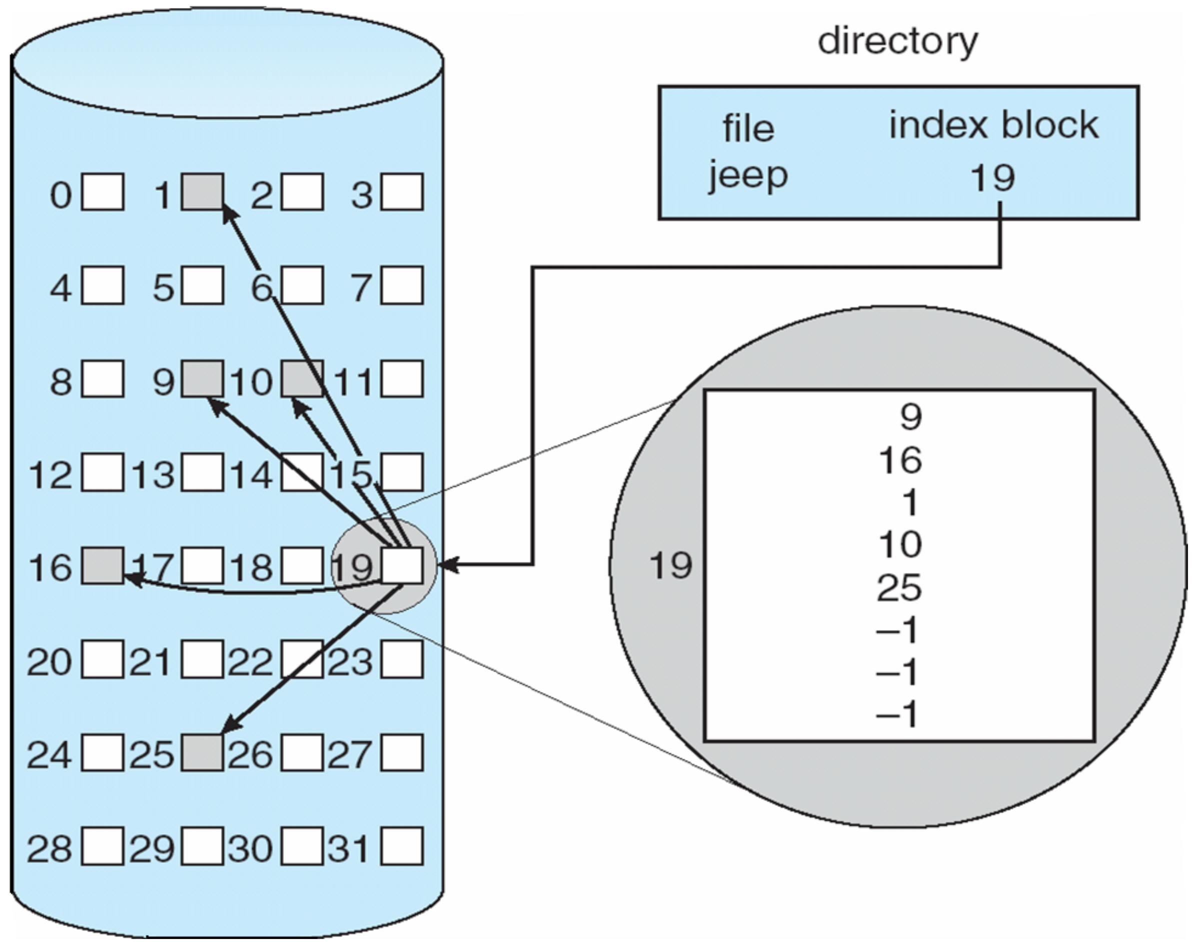



File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| ••• | ••• | ••• |
| File B | 1 | 5 |
| ••• | ••• | ••• |

# Indexed Allocation

- A directory entry points to a disk block that contains an index table for a file



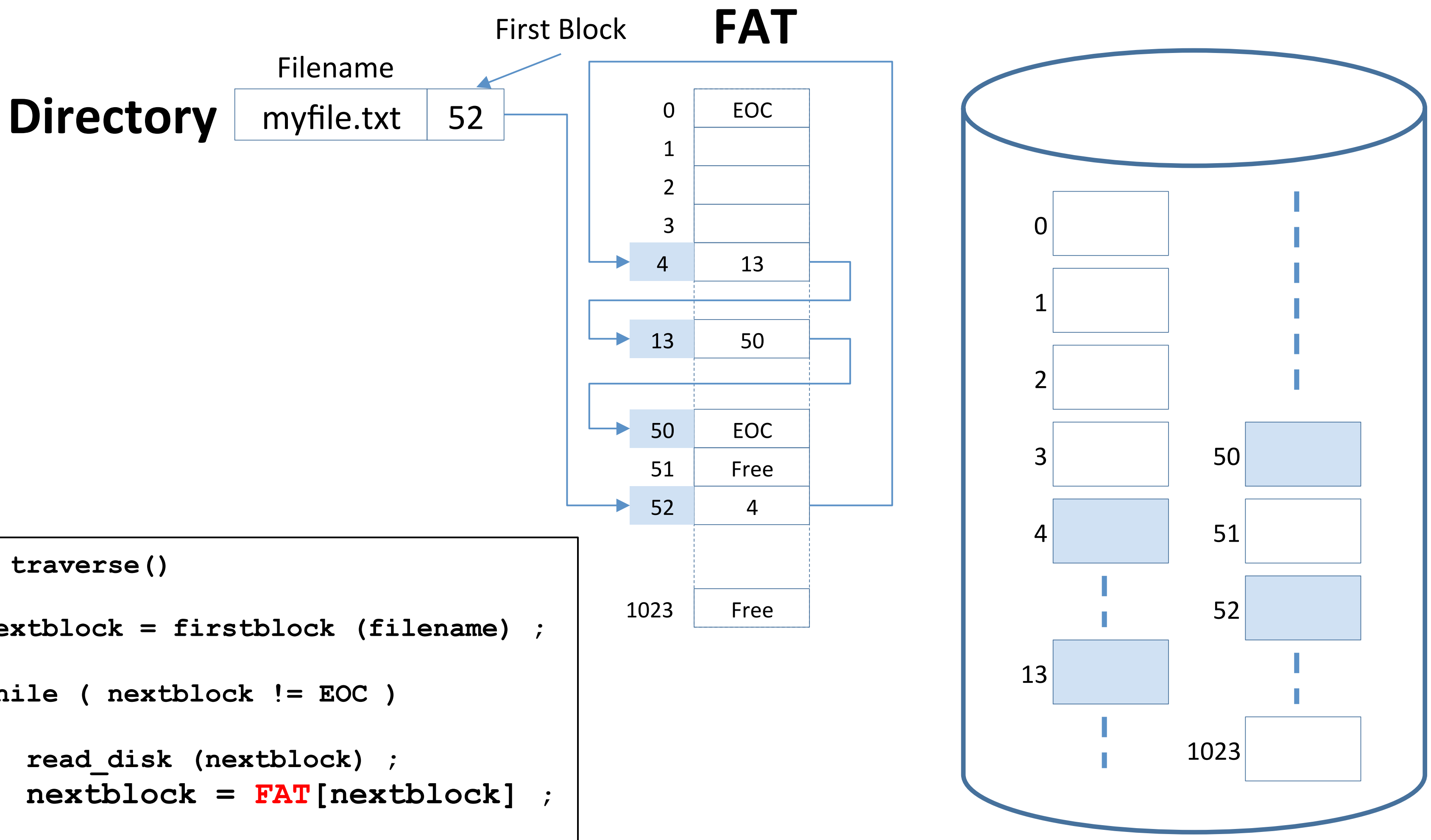
# Indexed Allocation

- Eliminates disadvantages of chained allocation
  - takes the pointers out of the data disk blocks and collects them in an extra index table
  - This table itself occupies one or more disk blocks
- Two important proponents
  - File Allocation Table FAT (MSDOS)
  - i-Nodes (Unix)

# File Allocation Table FAT

- Combines chained allocation with a separate index table – the “File Allocation Table” (FAT)
  - takes the pointers out of the disk blocks and collects them it in an extra table – the File Allocation Table (FAT)
- FAT table itself is stored at the beginning of the disk, occupies itself a couple of blocks
- Advantage:
  - FAT can be traversed very fast for block chains
  - Good for direct access to a single block as well as a sequential read of a file
  - A single block is available for data in its completeness
- Disadvantage
  - FAT itself may be large, has to be held in memory, must be saved on the disk as well
  - Size of FAT depends on size of hard disk: for each disk block there is an entry in the FAT

# File Allocation Table FAT




```
void traverse()
{
   nextblock = firstblock (filename) ;

   while ( nextblock != EOC )
   {
      read_disk (nextblock) ;
      nextblock = FAT[nextblock] ;
   }
}
```

# File Allocation Table FAT

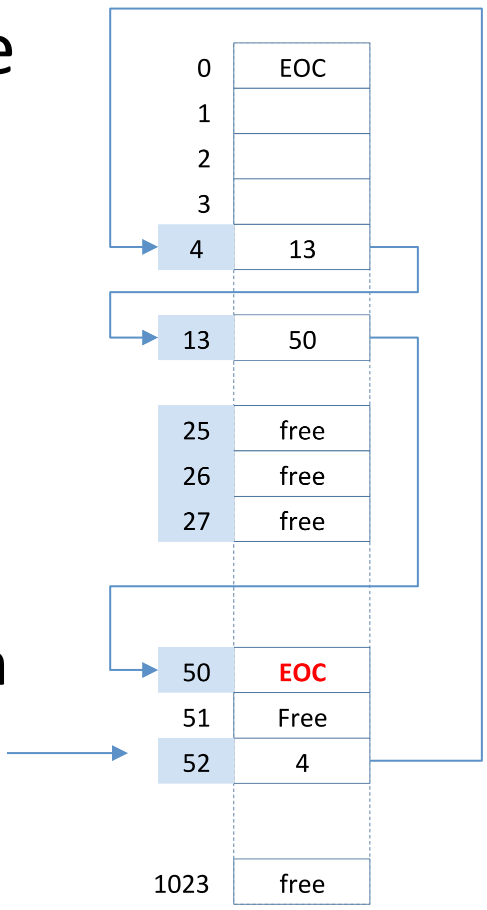
```
void traverse()
{
   nextblock = firstblock (filename) ;

   while ( nextblock != EOC )
   {
      read_disk (nextblock) ;
      nextblock = FAT[nextblock] ;
   }
}
```

# File Allocation Table FAT

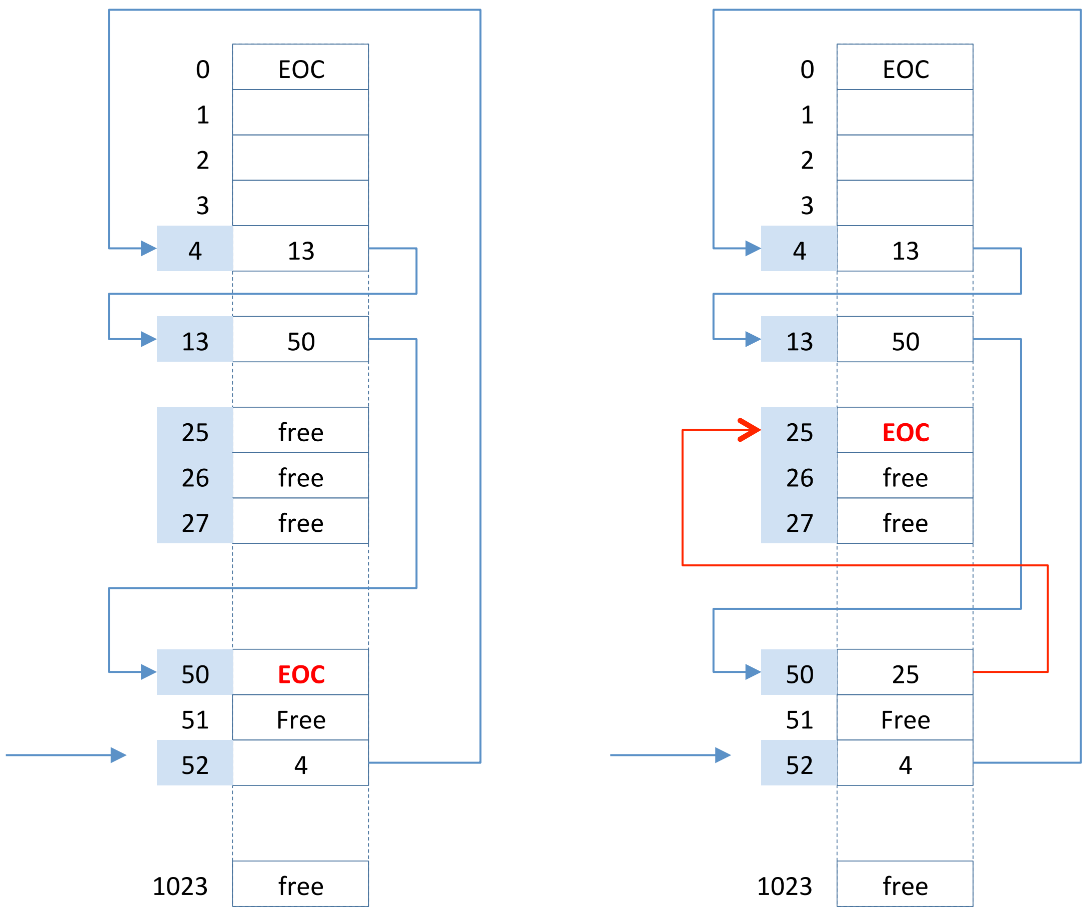
- File Allocation Table is loaded into memory, when disk is mounted by operating system
  - All chain pointers now in main memory
  - can easily be followed to find a block address
  - I/O action only needed to load actual disk block
- Entries in FAT form a block chain for a file
  - The index of the FAT entry is the block address of a file
  - The content of the FAT entry is the index of the next FAT entry in the chain and the block address of the next disk block of the file

# FAT: Extending a File

- Allocating a new disk block for a file
  - Information about free blocks are held in the FAT
- Find a FAT entry that is marked as “free” and extend the block chain
- I/O operation for FAT only:
  - As FAT is changed, it has to be written to disk – can be immediate or deferred

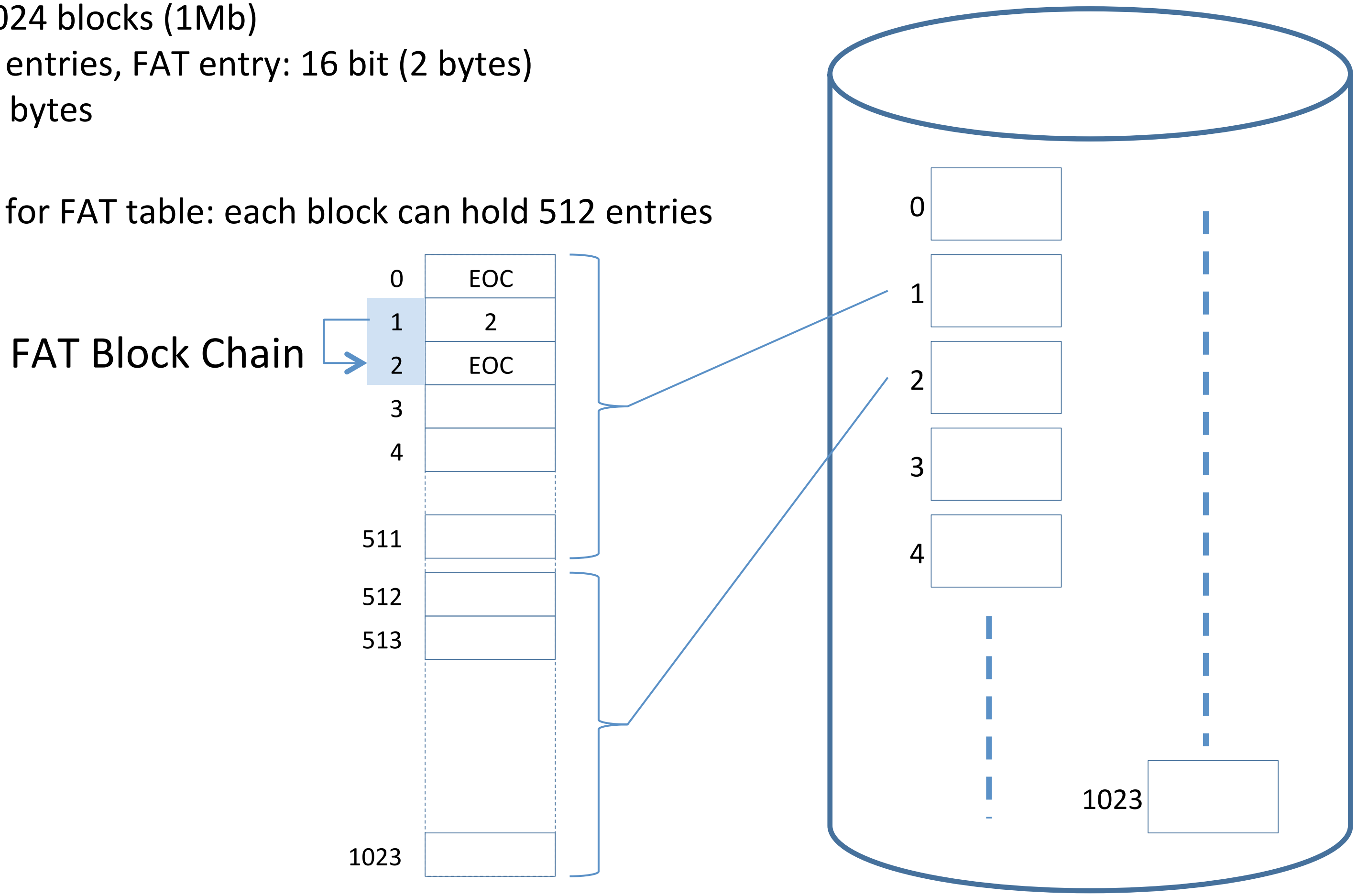


# FAT: Extending a File



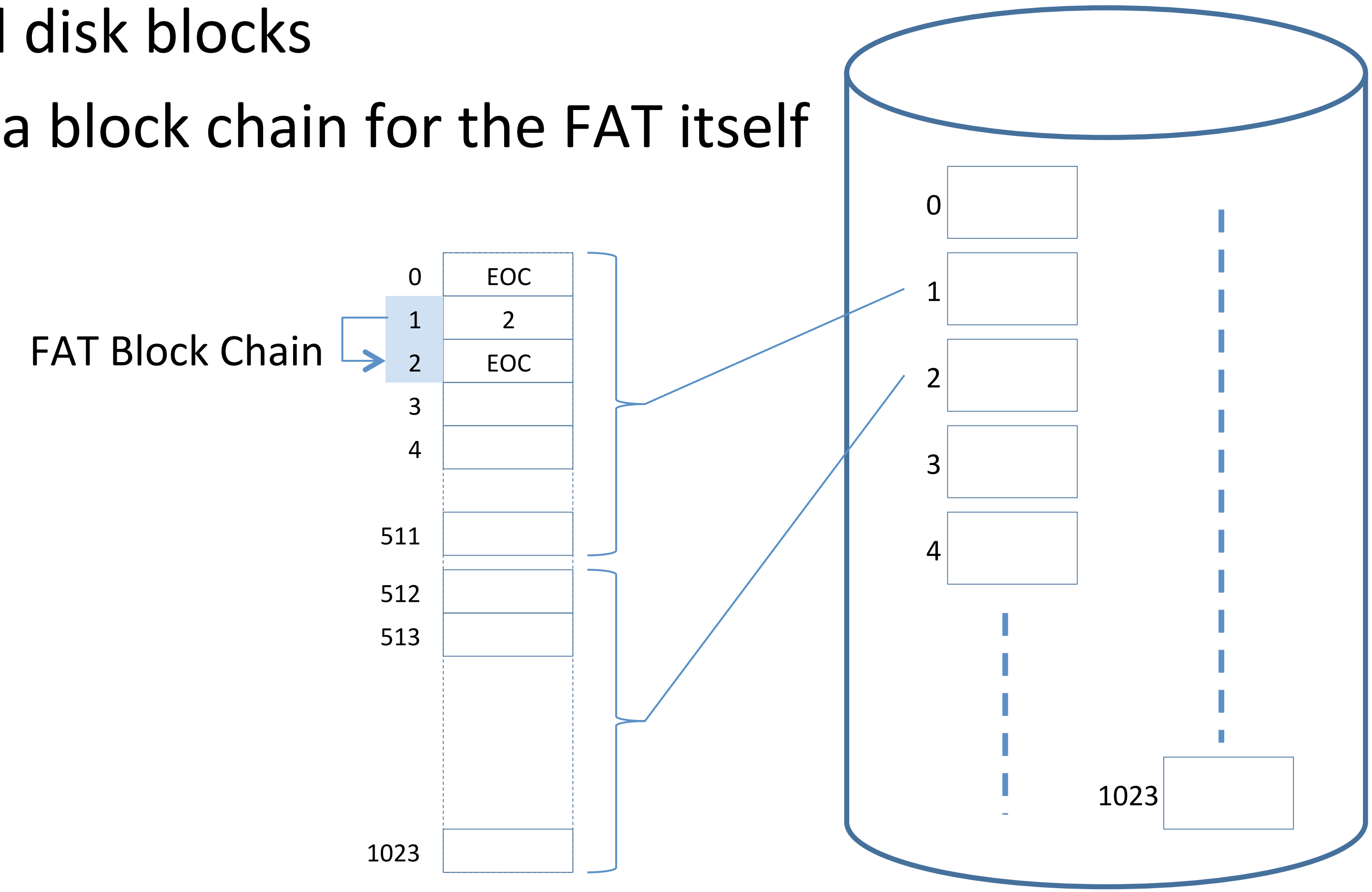
# Storing the FAT

- The FAT itself has to be permanently stored on disk
  - Will occupy disk blocks
  - There is a block chain for the FAT itself
- Given:
  - Hard disk has 1024 blocks (1Mb)
  - FAT table: 1024 entries, FAT entry: 16 bit (2 bytes)
  - Block size: 1024 bytes
- We need
  - Two disk blocks for FAT table: each block can hold 512 entries



# The FAT Block Chain

- Note:
  - The FAT itself occupies special reserved disk blocks
  - There is a block chain for the FAT itself



# Size of FAT

- Storage: 8GB USB drive, Block size: 4KB
- How many blocks do we need on the disk for the FAT?
- Remember:
  - 8GB = 8 x 1024 x 1024 x 1024 bytes
  - 4KB = 4 x 1024 bytes
- We calculate:
  - 8GB / 4KB = 8 x 1024 x 1024 x 1024 bytes / 4 x 1024 bytes = 2 x 1024 x 1024 = 2 Mio blocks
- Addressing:
  - We need at least $2^{21}$ entries in the FAT to address all 2 Mio blocks (2 x $2^{20}$)
  - We choose a 32-bit format for FAT entries (4 bytes), 1 block can hold 1024 entries: 4 x 1024 bytes / 4 bytes = 1024 entries
- Space for FAT on disk
  - 2 x 1024 x 1024 entries / 1024 entries = 2 x 1024 = **2048 blocks for the FAT**

| **Bytes** | **Exponent** | | | |
|---|---|---|---|---|
| **1,024** | $2^{10}$ | **1kb** | **1024bytes** | |
| **1,048,576** | $2^{20}$ | **1MB** | **1024kb** | **1024 x 1024** |
| **1,073,741,824** | $2^{30}$ | **1GB** | **1024MB** | **1024 x 1024 x 1024** |
| **4,294,967,296** | $2^{32}$ | **4GB** | **4 x 1024MB** | **4 x 1024 x 1024 x 1024** |
| **1,099,511,627,776** | $2^{40}$ | **1TB** | **1024GB** | **1024 x 1024 x 1024 x 1024** |
| **1,125,899,906,842,620** | $2^{50}$ | **1PB** | **1024TB** | **1024 x 1024 x 1024 x 1024 x 1024** |
| **1,152,921,504,606,850,000** | $2^{60}$ | **1EB** | **1024PB** | **1024 x 1024 x 1024 x 1024 x 1024 x 1024** |
| **18,446,744,073,709,600,000** | $2^{64}$ | **16EB** | | **16 x 1024 x 1024 x 1024 x 1024 x 1024 x 1024** |

# FAT: Deleting Files

- Deleting a file is fast
- Two actions
  - The directory entry for a file is marked as deleted
    - First character of filename is set to some non-printable value to make it “invisible” (in the FAT implementation, it is set to 0xE5)
  - All entries of the block chain are set to “free”
- I/O operation for FAT only:
  - As FAT is changed, it has to be written to disk – can be immediate or deferred

# Free Space Management FAT

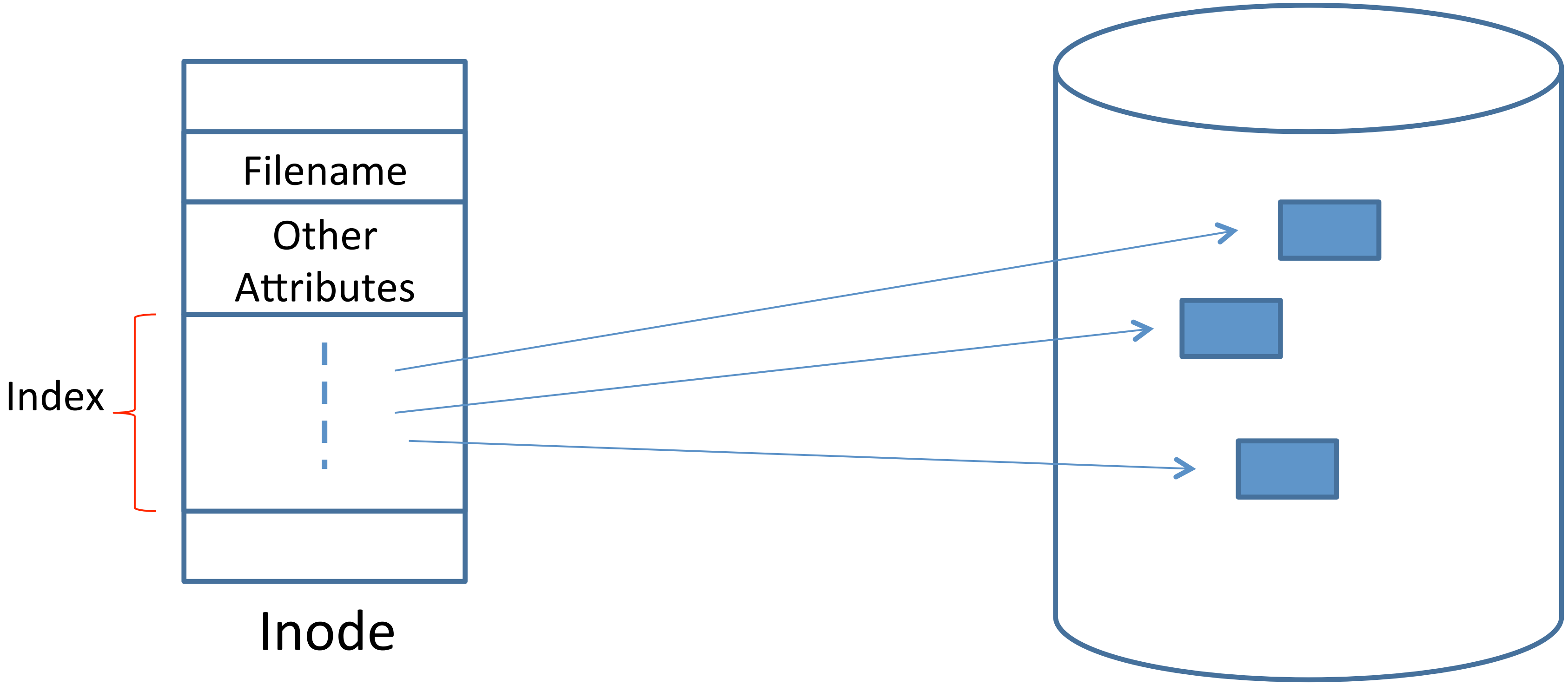
- Information in FAT table determines whether a block on disk is free
  - All free blocks are marked as "unused"
- When a file is deleted
  - The directory entry for a file is marked as deleted
    - First character of filename is set to 0xE5
  - The block chain for this file is cleared in the FAT
    - All FAT entries of such a chain are set to a value indicating that it is "unused"
- Blocks on disk are untouched, no update of their content is needed, no I/O operations

# Indexed Allocation: i-Nodes

# i-Nodes

- All types of Unix files are managed by the operating system by means of i-Nodes
  - Is a control structure ("index" node) that contains the key information needed by the operating system for a particular file
    - Describes its attributes
    - Points to the disk blocks allocated to a file
- The i-Node is an index to the disk blocks of a file
  - One i-Node per file
  - The i-Node records only the blocks allocated to a file
- Requires a management of a separate list of free blocks

# i-Nodes




- A simple list of block references (single-level) allows fast access to all blocks of a file
- But: it restricts the maximum size of a file

# Hierarchical Index

- i-Node manages n-level hierarchical index
  - One disk block can only contain a small list of addresses to disk blocks
  - Has therefore multiple levels: an entry may point to a sub-index table
    - Entries in the i-Node points to blocks on disk that contain pointers to other blocks
- Can address very large files

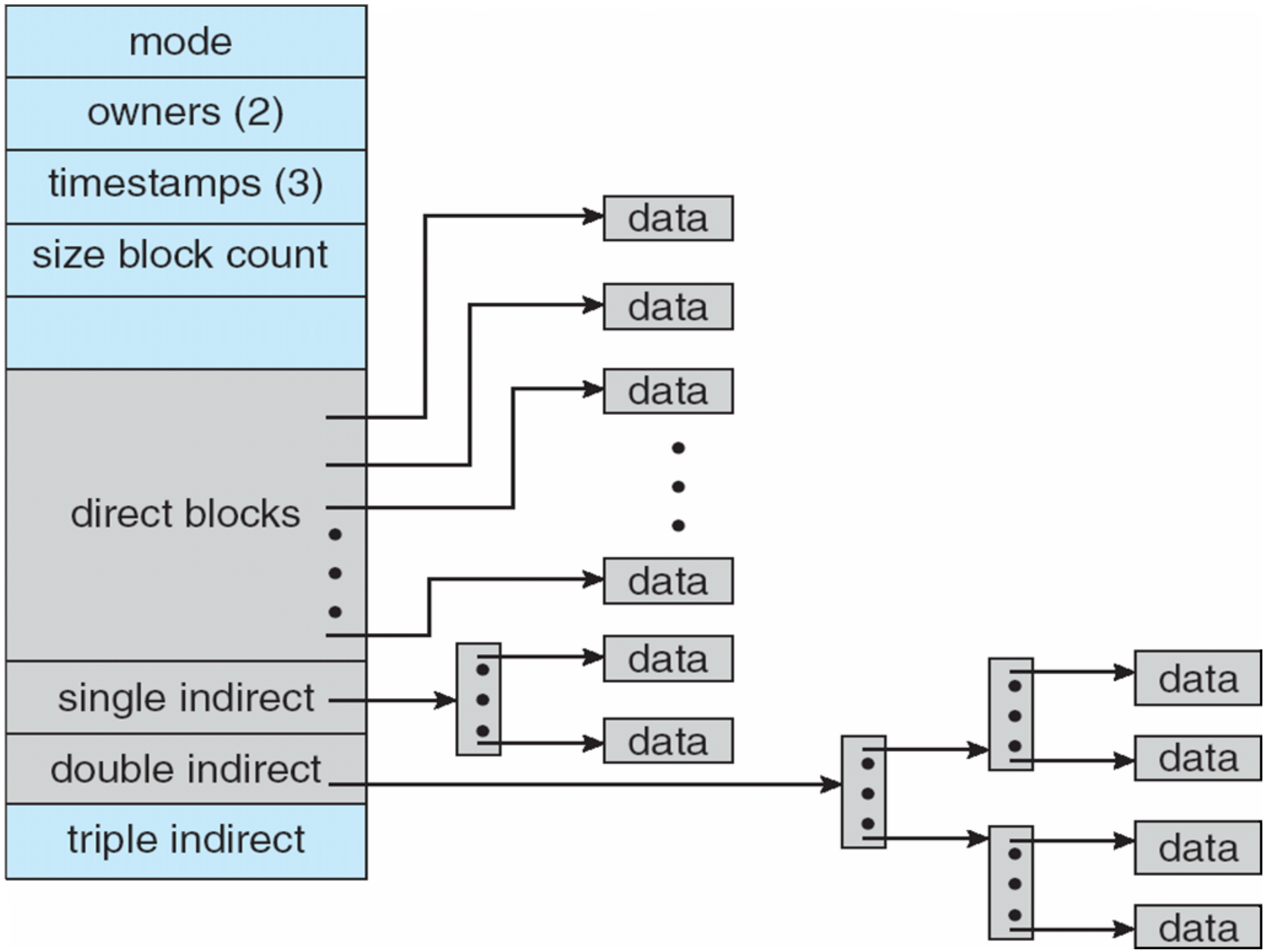
# Indexed Allocation: i-Nodes

- How can we distinguish between index blocks and data blocks?
- How do we know how many levels the index has?
- i-Node contains two different versions of index entries
  - Direct block index entry
  - Indirect block index entry

# Direct and Indirect Referencing

- First N index entries point directly to the first N blocks allocated for the file
- If file is longer than N blocks, more levels of indirection are used
- Inode contains three index entries for “indirect” addressing
  - “single indirect” address:
    - Points to an intermediate block containing a list of pointers
  - “double indirect” address:
    - Points to two levels of intermediate pointer lists
  - “triple indirect” address:
    - Points to three levels of intermediate pointer lists

# i-Node Indexed References of Disk Blocks



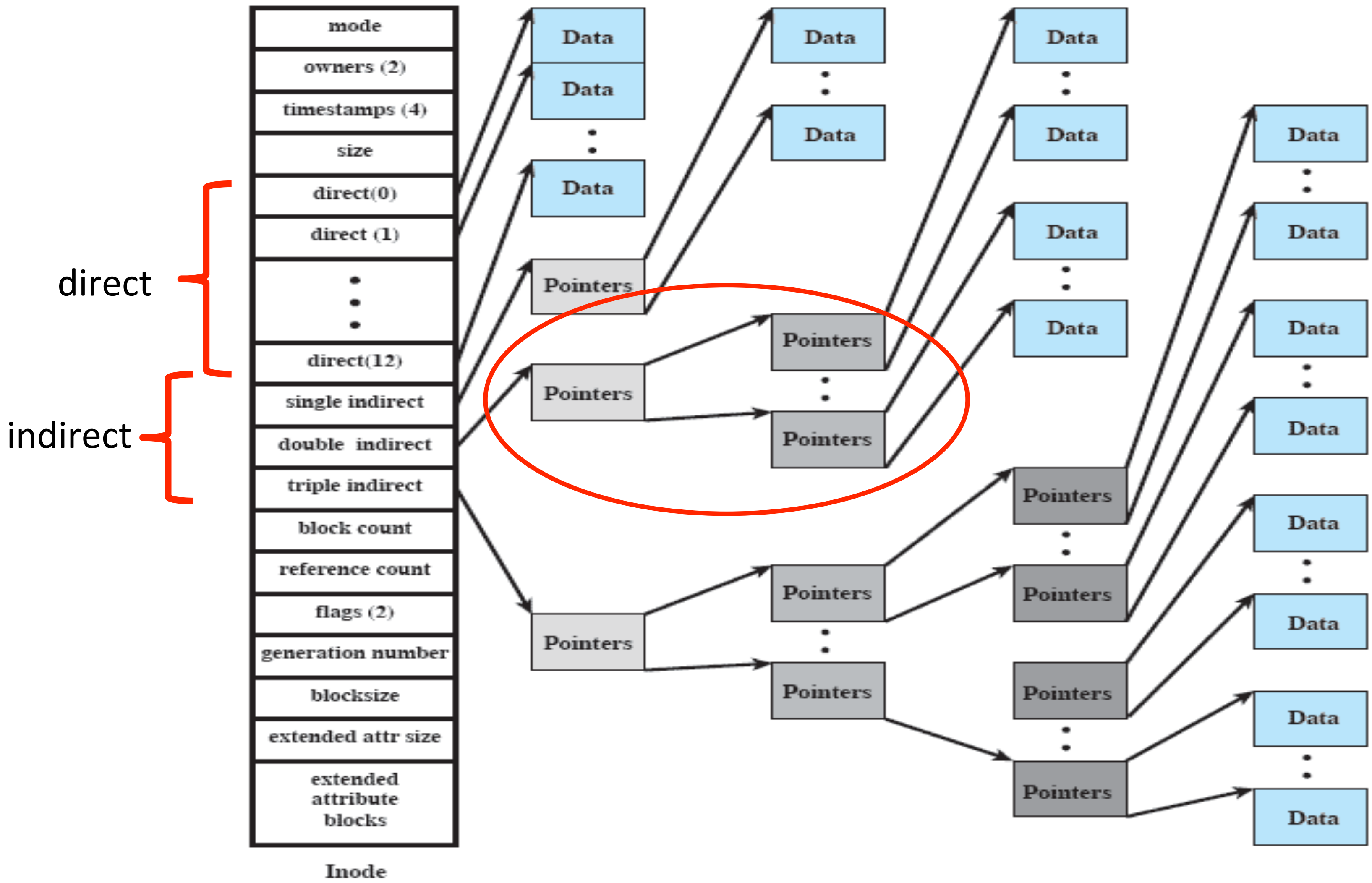
# i-Node Direct and Indirect Indexing

- Example implementation with 13 index entries:
  - i-Node contains a list of 13 index entries that combine four different forms of index
  - Direct block references:
    - 10 entries of this list point directly to file data blocks
  - Single indirect (two levels):
    - Entry 11 is regarded as always pointing to an index disk block: this index block contains address of actual file data blocks
  - Double indirect (three levels): entry 12 is regarded to be the starting point of a three-level index
  - Triple indirect (four levels): entry 13 is regarded to be the starting point of a four-level index

# i-Node Direct and Indirect Indexing

- Based on which entry in the i-Node is used, the file system management can distinguish whether an indexed block is a data block or another level of one of the indices
- Assumption
  - There are many small files, the number of directly referenced blocks may be enough
  - For larger files, the additional indices are used

# File Allocation with i-Node



# i-Node Table

- Operating system has to manage the i-Node table
  - When a file is opened / created, its i-Node is loaded into the i-Node table
  - The size of this table determines the number of file that can be held open at the same time

# File Allocation with i-Nodes

- What is maximum size of a file that can be indexed:
  - Depends of the capacity of a fixed-sized block
- Example implementation with 15 index entries:
  - 12 direct, single (13) / double (14) / triple (15) indirect
  - Block size 4kb, holds 512 block addresses (32-bit addresses)

| Level | Number of Blocks | Number of Bytes |
|---|---|---|
| **Direct** | 12 | 48K |
| **Single Indirect** | 512 | 2M |
| **Double Indirect** | $512 \times 512 = 256K$ | 1G |
| **Triple Indirect** | $512 \times 256K = 128M$ | 512G |

# i-Nodes

- Advantage
  - i-Node is only loaded into memory when a file is opened
  - Good for managing very large disks efficiently
  - We need a list of i-Nodes of open files: size of this list determines how many files may be open at the same time
- Disadvantage
  - The i-Node only has a fixed list for block references
  - If a file is small, fast and efficient management
  - If file is large, the i-Node has to be extended with a hierarchy of indirect block lists connected to the i-Node, needs extra I/O operations to scan the index